Non-Race Concurrency Bug Detection Through Order-Sensitive Critical Sections

Ruirui Huang, Erik Halberg, G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
{rh335,esh64,gs272}@cornell.edu

ABSTRACT

This paper introduces a new heuristic condition for non-race concurrency bugs, named order-sensitive critical sections, and proposes a run-time bug detection scheme based on the condition. The order-sensitive critical sections are defined as a pair of critical sections that can lead to non-deterministic shared memory state depending on the order in which they execute. In a sense, the order-sensitive critical sections can be seen as extending the intuition in using data races as a potential bug condition to capture non-race bugs. Experiments show that the proposed scheme provides a good coverage for multiple types of non-race bugs, with a small number of false positives. For example, the scheme detected all 9 real-world non-race bugs that were tested as well as over 90% of injected non-race bugs. Additionally, this paper presents an efficient hardware architecture that supports the proposed scheme with minor hardware changes and a small amount of additional state - a 9-KB buffer per core and a 1-bit tag per data cache block. The hardware-based scheme could still detect all 9 real-world bugs that were tested and more than 84% of the injected non-race bugs. Moreover, the hardware supported scheme has a negligible impact on performance, with a 0.23% slowdown on average.

1. INTRODUCTION

As computing hardware moves to multi-core and many-core systems, future software needs to be parallelized in order to benefit from the increasing computing resources. However, writing a correct parallel program is notoriously difficult, partly because of non-determinism in concurrent program executions. Because thread executions can be interleaved in many ways, a parallel program may produce a non-deterministic outcome even for identical program inputs if threads are not properly synchronized. Such a non-deterministic behavior, if not intentional, is often referred to as a concurrency bug.

In this paper, we propose a new concurrency bug detection scheme that is designed to detect common non-race bugs by extending the intuition behind traditional data races into critical sections. A data race refers to conflicting memory accesses - accesses from multiple threads to the same location with at least one write - without any synchronization. Data races are commonly considered as potential concurrency bugs because the conflicting accesses can execute in an arbitrary order and result in different memory state among multiple runs. Similarly, we observe that ordering of certain critical sections may change from run to run, introducing non-determinism in memory state. We call such critical sections as order-sensitive critical sections.

This notion of *order-sensitive critical sections* provides a new condition for detecting potential concurrency bugs. However, unlike traditional data races, there are many cases when critical sections do not introduce non-determinism. Therefore, the question is whether one can effectively distinguish order-sensitive critical sections, which are likely to indicate a bug, from other legitimate uses of critical sections. We solve this challenge by studying how programmers typically use critical sections without introducing non-determinism, and developing a set of heuristic conditions to filter out such legitimate cases including explicit ordering, data parallel operations, redundant writes, and commutative operations.

The paper presents a run-time algorithm, named OSCS, that uses the notion of order-sensitive critical sections to detect real-world concurrency bugs. The algorithm relies on vector clocks similar to the ones used for data race detection [6, 21] to keep track of ordering restrictions and adds extensions to filter out critical sections whose results do not depend on the execution order.

In practice, we found that this new approach can detect a broad range of non-race concurrency bugs with minimal false positives. In our experiments, the OSCS detection scheme flagged all 9 real-world non-race bugs that we could find and test, including atomicity violation, ordering violation, and multi-variable bugs. The scheme also detected most (91%) of randomly injected atomicity and ordering violation bugs in PARSEC and SPLASH benchmarks. Moreover, experiments on Apache, Aget, Pbzip2, MySQL, Mozzila, SPLASH2, and PARSEC suggest that this new approach only introduces a small number of false positives.

As an additional optimization, this paper also shows that the amount of meta-data can be significantly reduced by using scalar timestamps instead of vector clocks and keeping them only for shared memory locations. This optimization has a minimal impact on detection capability as long as vec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA '13 Tel-Aviv, Israel
Copyright 2013 ACM ACM 978-1-4503-2079-5/13/06 ...$15.00.

tor clocks are maintained for critical ordering constraints. The idea of only keeping scalar timestamps is similar to a recent software optimization (FastTrack) [7] and improves scalability. More importantly, the reduction in meta-data enables an efficient hardware implementation of the OSCS algorithm with a small amount of additional state: a 9-KB buffer per core and a 1-bit tag per data cache block. Experimental results show that the hardware OSCS implementation can still detect all 9 real-world non-race bugs tested and more than 84% of the injected non-race bugs. Moreover, the experimental results show that the hardware supported scheme has a negligible impact on performance, with a 0.23% slowdown on average.

While a number of techniques have been proposed to detect non-race concurrency bugs, the proposed approach represents a unique contribution. The OSCS scheme can detect multiple types of bugs including atomicity violation, ordering violation, and multi-variable bugs rather than focusing on a single type. Also, OSCS does not require training or application-specific information as it relies on a bug condition that is common across programs. However, the proposed technique is still a heuristic and there is no guarantee on bug detection coverage. In this sense, the proposed scheme complements an existing body of work in non-race bug detection.

The following summarizes the main contributions:

- *The notion of order-sensitive critical sections*: We introduce a new condition that can serve as a general indication of non-race concurrency bugs along with heuristics to check the condition.
- *Run-time detection algorithm*: We present a new algorithm that can effectively detect non-race bugs based on the notion of order-sensitive critical sections. We also introduce a set of optimizations to make the algorithm more scalable and amenable to hardware implementations.
- *Efficient architecture support*: We present an effective and efficient hardware implementation of the OSCS algorithm, which has minimal performance overhead and still has a high detection coverage.

The rest of the paper is organized as follows. Section 2 presents the idea of detecting non-race concurrency bugs based on order-sensitive critical sections. Then, Section 3 describes how this idea can be realized as a detection algorithm, and Section 4 presents a hardware implementation of the algorithm. Section 5 evaluates the proposed scheme. Section 6 discusses related work, and Section 7 concludes the paper.

2. ORDER-SENSITIVE CRITICAL SECTION

This section introduces a new condition for non-race concurrency bugs. We first present assumptions that we make, and discuss how traditional data races capture concurrency bugs. Then, we extend the intuition into non-race bugs.

2.1 Assumptions

The proposed approach makes a couple of assumptions that are common across many concurrency bug detection schemes. First, this work assumes that programs use the shared memory model. Except for creating a thread and waiting for a termination, threads communicate through accesses to shared memory locations.

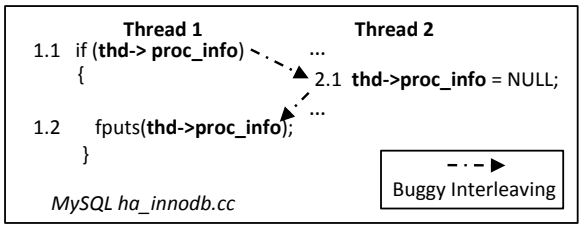


Figure 1: A data race example (from MySQL [10]).

We also assume that synchronization operations that control interleaving patterns among threads can be explicitly identified. Programmers often use a library such as Pthreads for synchronization operations. In such cases, synchronization operations can be easily identified from the library calls. If a programmer uses custom synchronization primitives, our approach assumes that such primitives can be either explicitly marked by the programmer or automatically identified. For example, previous studies show that primitives such as spinlocks can be automatically detected [20, 22].

In general, parallel programs rely on two types of synchronization primitives to control thread interleaving. Primitives such as barriers and wait-signal pairs explicitly enforce a predetermined ordering among threads. In essence, the synchronization makes the thread interleaving deterministic. In this paper, we refer to these primitives as *ordering* synchronization operations. On the other hand, primitives such as mutex and semaphores provide mutually exclusive code regions, often called *critical sections*, without enforcing a particular execution order. Thus, such critical sections can execute in a non-deterministic order in each run. We will refer to such primitives as *mutex* synchronization operations.

2.2 Bug Detection Through Data Races

Informally, concurrency bugs can be considered as mistakes in synchronization that allow an unintended thread interleaving pattern, which results in inconsistent program outcomes from run to run even for identical inputs. Programmers may intentionally allow non-deterministic outputs to improve performance when accuracy is not critical (logging, statistics, etc.), However, non-deterministic outputs are infrequent in practice and often indicate a bug.

Because exhaustively testing outputs for non-determinism is infeasible in practice, concurrency bug detection schemes often check for improper synchronization patterns, which lead to non-deterministic memory state. For example, data races capture a large class of concurrency bugs where a synchronization operation is missing between a pair of conflicting memory accesses. Here, conflicting accesses are defined as ones from different threads to the same memory location, with at least one write. The data races imply that conflicting accesses can be executed in an arbitrary order, resulting in non-deterministic shared memory state.

Figure 1 shows a data race in MySQL that results in an atomicity violation bug. In this example, none of the accesses to the shared pointer `thd->proc_info` is protected by synchronization. These accesses can be freely reordered, resulting in a fault if the pointer is set to be NULL by *2.1* between *1.1* and *1.2*. To prevent the data race, both *1.1-1.2* and *2.1* need to be need to be protected by a critical section to ensure an atomic execution.

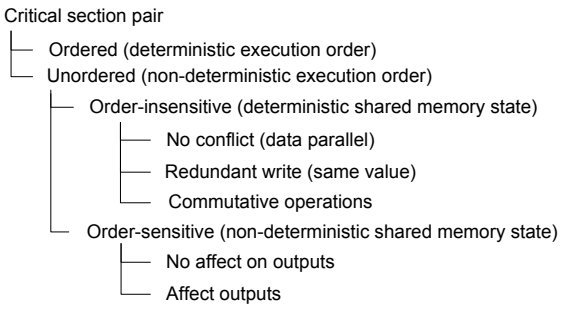


Figure 2: Types of critical section pairs.

2.3 Non-Determinism in Critical Sections

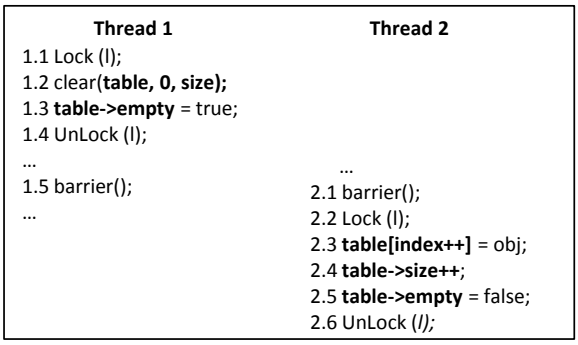
While data races can detect a broad range of concurrency bugs where conflicting memory accesses are not controlled at all, recent studies show that many concurrency bugs do not fall into data races [10]. Programmers can remove data races by placing shared memory accesses within critical sections. However, because the critical sections can still execute in an arbitrary order, program outputs may still be non-deterministic even without data races.

In this paper, we aim to develop a general technique to detect a broad range of non-race concurrency bugs. To achieve this goal, we extend the intuition behind data races, where non-deterministic shared memory state from the lack of synchronization is used as an indicator for a concurrency bug, to critical sections. In essence, we look at a pair of critical sections with conflicting memory accesses and use it as an indicator for a concurrency bug if the two critical sections can lead to non-deterministic shared memory state depending on the order in which they execute. We call such critical section pairs as *order-sensitive critical sections*.

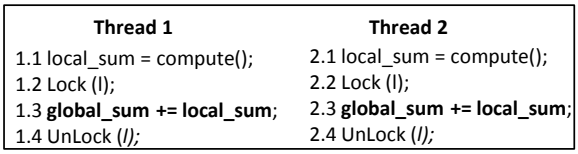
To understand when a critical section pair leads to non-deterministic shared memory state, let us consider how critical sections are typically used to produce deterministic state even when the mutex synchronization does not enforce a specific ordering between critical sections. Figure 2 categorizes common relationships between a pair of critical sections.

A critical section pair can first be categorized based on whether the two critical sections can truly run in an arbitrary order. Even though the mutex synchronization does not enforce any ordering, programmers may use an additional ordering synchronization operation to ensure a certain order between critical sections. In this case, the critical sections execute in a deterministic order. For example, Figure 3 (a) shows an *ordered* critical section pair from a simple database table manipulation example. In the example, one thread (Thread 1) initializes the table to an empty state before reaching a barrier, and another thread (Thread 2) can add entries to the table only after the barrier. Therefore, the critical section in Thread 1 is ordered to execute before the critical section in Thread 2 by the barrier ordering synchronization operation.

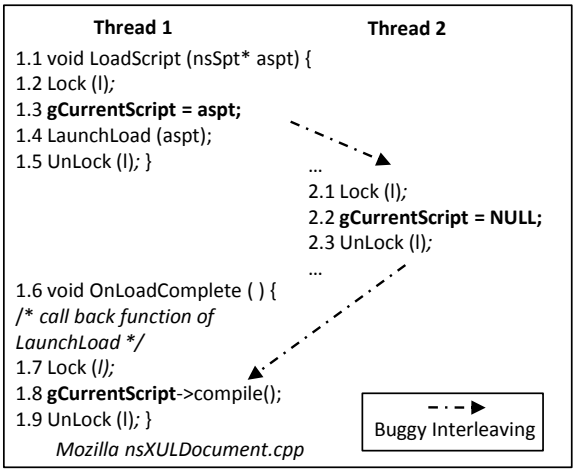
An unordered critical section pair can either be order-insensitive or order-sensitive. The order-insensitive pairs imply that the shared memory state after the two critical sections will be identical no matter which order they run. There are a few common cases where the ordering of critical sections does not affect the program state. First, critical sections may work on disjoint sets of memory locations without any conflicting accesses. Such a case is common for a data-parallel part of a program. Second, a conflicting write may be redundant, resulting in the same value in memory no matter which order the two critical sections execute. For example, a program may initialize a variable in multiple threads to a same value without synchronization. Finally, the operations in the critical sections may be commutative. For example, Figure 3 (b) shows an example where two threads add a partial sum to the global sum. Because an addition is commutative, the result is identical no matter which order the two threads run.



(a) Ordered critical section pair.



(b) Commutative critical section pair.



(c) Order-sensitive critical section pair. Non-race atomicity violation bug (Mozilla-1) [11].

Figure 3: Examples for different types of critical section pairs.

If the execution order of two critical sections affect the resulting shared memory state, the critical section pair is *order-sensitive*. In this case, the shared memory state can be non-deterministic even for identical inputs. If the inconsistency in memory affects a program output, such order-sensitive critical sections represent concurrency bugs. For example, Figure 3 (c) shows an atomicity violation example from Mozilla [11]. In this case, each memory access to the shared variable `gCurrentScript` is protected by a lock. However, the value of `gCurrentScript` depends on the order in which the two threads' critical sections execute. The

program can still crash when the thread interleaving follows *1.3 - 2.2 - 1.9*; `gCurrentScript` will be NULL for *1.9*.

In this work, we propose to use the order-sensitive critical sections as an indicator for bugs. In that sense, we refer to our detection scheme as OSCS (Order-Sensitive Critical Sections). Note that we only consider *shared* memory locations, which are used by multiple threads, to determine order-sensitivity. We do not use non-determinism in thread local state as an indicator for bugs. This is because non-deterministic shared memory state is much more likely to result in non-deterministic program outputs compared to non-deterministic local state. For example, in a case where multiple consumers work on a set of data in parallel, the thread local state will depend on which data that a particular thread works on and can easily be different from run to run even when outputs are deterministic.

2.4 Detection Heuristic

Here, we discuss how to detect order-sensitive critical sections in practice. Instead of precisely detecting all order-sensitive critical sections, the goal is to develop a simple heuristic that detects most order-sensitive critical sections with *minimal false positives*. Also, we want the heuristic to be simple enough for run-time checks. Just like data race detectors are often used even when they cannot detect all concurrency bugs, we believe a detection scheme will be useful if it covers a broad range of bugs with low false positives.

While there can be many ways to detect order-sensitive critical sections, our approach defines the order-sensitive critical sections indirectly as critical sections that are neither ordered nor order-insensitive. Then, the approach relies on a set of heuristics to filter out common patterns for ordered and order-insensitive critical sections as shown below. The detailed run-time detection algorithm is explained in the next section.

Ordered critical sections: The scheme keeps track of restrictions from all ordering synchronization operations. If there exists direct or indirect ordering restriction between two critical sections, the pair is considered to be ordered.

No conflict (order-insensitive): If there is no conflicting accesses between two critical sections, the pair is order-insensitive. This condition can be checked by keeping track of previous accesses for each memory location.

Redundant writes (order-insensitive): If a conflicting write does not change the value in memory, the write operation is redundant. This condition can be checked by comparing the memory value before and after a write.

Commutative operations (order-insensitive): Precisely checking if two critical sections are commutative requires understanding of detailed semantics of operations as well as data dependence. Instead, we use a simple heuristic based on a memory access pattern to conservatively filter out commutative operations. Intuitively, in order for an operation on a shared memory location to be commutative, each thread needs to first read the current value before updating it, often atomically. For example, adding a partial sum to a global sum in Figure 3 (b) requires reading the current global sum and writing the updated sum within a critical section.

Based on this intuition, our heuristic considers a pair of critical sections to be commutative if both contain a read followed by a write for each memory location with conflicting accesses between two critical sections. We call the sequence

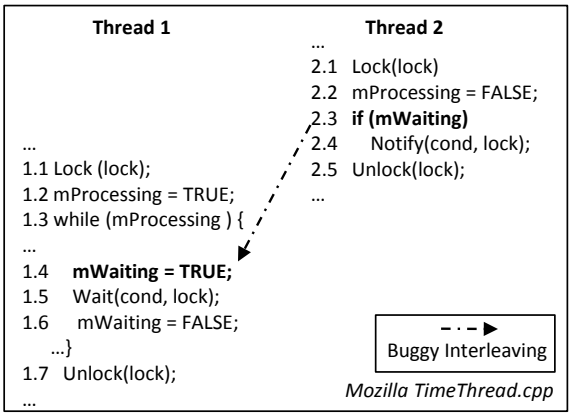


Figure 4: An ordering violation bug example (KB3) [24].

of a read followed by a write to the conflicting location a *read-write* sequence.

To understand how the proposed checks can detect bugs, let us consider the atomicity violation example in Figure 3 (c). In this example, the critical sections in Thread 1 (*1.2 to 1.5*) and Thread 2 (*2.1 to 2.3*) are unordered and have conflicting writes to `gCurrentScript`. Also, each critical section only contains one write instead of a read-write sequence, which indicates that the critical sections are not commutative. Therefore, the two critical sections will be flagged as order-sensitive.

As another example, Figure 4 shows an ordering violation example from Mozilla. The program hangs if the thread interleaving follows *2.3 - 1.4* because `mWaiting` would be `false` at *2.3* and Thread 1 waits at *1.5* for a notification that will never be sent. The critical sections in this example will be flagged as order-sensitive because they are not ordered, and have conflicting accesses to `mProcessing` and `mWaiting` without a read-write sequence in both critical sections.

2.5 Limitations

The proposed detection scheme, OSCS, is a heuristic for detecting non-race concurrency bugs. In that sense, OSCS can have both false negatives and false positives, like any other heuristic methods. For example, data race detectors cannot detect all concurrency bugs (false negatives) and may also incorrectly identify programmer intended races for bugs (false positives). Our experiments, however, indicate that OSCS can detect a broad range of bugs with a small number of false positives.

From the false positive perspective, OSCS relies on the assumption that non-deterministic shared memory state indicates a bug. However, if the non-deterministic shared memory state does not lead to non-deterministic outputs or the outputs are intentionally allowed to be non-deterministic, our approach can lead to a false positive. The OSCS scheme also uses a heuristic to detect commutative critical sections where *read-write* sequences in both critical sections are considered as an indicator for a commutative operation. This heuristic can lead to false positives when it is not enough to detect all commutative operations as shown in Figure 5. In this example, each thread is executing the same piece of code, which updates the global maximum error value. A false positive happens when either Thread 1 or Thread 2's `local_err` is less than global variable. This results in one of

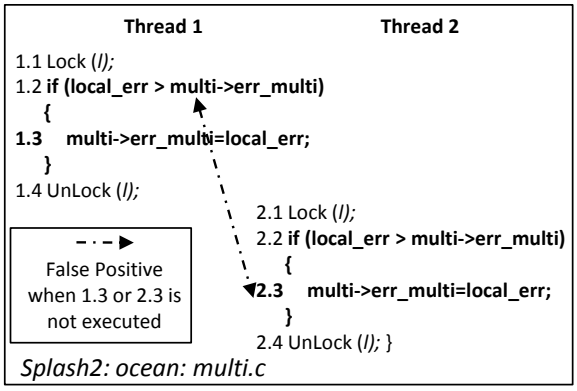


Figure 5: A false positive example from Ocean.

the critical sections with only a read, while the other critical section has a read followed by a write to the global variable.

There are a couple of approximations in the proposed heuristics that can lead to false negatives (missed bugs). First, we do not detect non-deterministic thread local state as a bug because they are difficult to check and could lead to false positives. Second, the heuristic to detect commutative critical sections can be conservative and incorrectly mark non-commutative ones as commutative. In addition to these approximations in our heuristics, OSCS is designed to detect bugs in accesses within critical sections, and does not detect data race bugs or bugs from misplaced ordering synchronization operations.

Note, however, that false positives and negatives are common for any heuristic method. An important question is how well the heuristic works in practice. Our experiments on real-world application programs including Apache, MySQL, Aget, Pbzip2, PARSEC, and SPLASH2 indicate that there are only a small number of false positives. The experiments on real-world and injected bugs show that the OSCS scheme is quite effective in detecting non-race bugs; the scheme detected all real-world bugs and over 90% of injected bugs.

3. OSCS DETECTION ALGORITHM

This section presents the run-time algorithm for detecting order-sensitive critical sections. We first present a high-level overview of the algorithm and the meta-data, followed by the detailed algorithm and optimizations.

To be general, we describe the algorithm using *release* and *acquire* instead of individual synchronization operations. Synchronization primitives can be considered as acquiring and releasing tokens, which we refer to as *synchronization objects*. For example, mutual exclusion requires each thread to acquire a token (lock) before entering a critical section and to release the token after the critical section. Similarly, barriers can be realized by having each thread to release a token upon reaching the barrier and waiting to acquire tokens from all other threads before proceeding.

3.1 Overview

In a high-level, the OSCS algorithm detects a potential concurrency bug by detecting two critical sections with conflicting memory accesses and checking if the pair is order-sensitive from the perspective of that shared location.

Figure 6 shows an overview of the OSCS algorithm on each memory access. The algorithm first checks if there is a

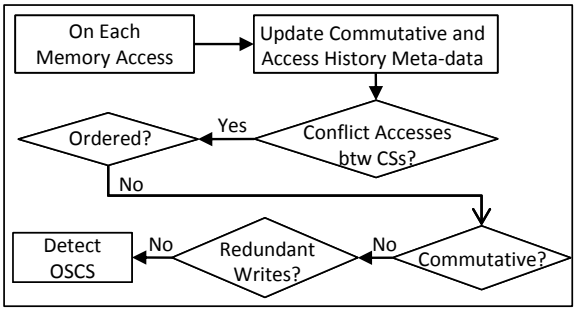


Figure 6: OSCS operations on every memory access.

Meta-data:
1. `CSList[TID]`:
   A set of critical section(s) that a thread is currently in.
2. `PrevReadCSList[Addr][TID]/PrevWriteCSList[Addr][TID]`:
   A set of critical section(s) that a thread was in when the most recent memory accesses happened in that thread.
3. `ThreadVClk[TID1][TID2]`:
   A vector clock per thread.
   A timestamp is stored in each element.
4. `PrevReadVClk[Addr][TID]/PrevWriteVClk[Addr][TID]`:
   Time stamp per address per thread for most recent read/write.
5. `OrderObjVClk[OrderObj][TID]`:
   A vector clock per ordering synchronization object.
   A timestamp is stored in each element.
6. `CSRWFlag[Addr][TID]`:
   A single bit flag per memory address per thread that indicates wether a read-write sequence exists within a critical section.
7. `DetectionList[TID]`:
   A list of potentially buggy memory access pairs in each thread.

Figure 7: Meta-data required for the OSCS algorithm.

previous access to the memory location from another thread, which forms a conflicting access pair with the current access. Then, the algorithm checks if both conflicting accesses are protected by a critical section with the same lock. These checks identify a critical section pair with conflicting memory accesses, which are not data races.

Once a critical section pair is identified, the algorithm checks if the pair is order-sensitive by filtering out ordered and order-insensitive ones. First, the algorithm checks if the critical sections are ordered by keeping track of constraints from ordering synchronization operations. Then, the algorithm checks whether the critical section pair is order-insensitive due to commutative operations or redundant writes. If the critical section pair is neither ordered nor order-insensitive, the algorithm logs it as a bug.

In order to perform the necessary checks, the algorithm maintains a set of meta-data on each memory accesses as well as each synchronization *acquire* or *release* operation. Figure 7 shows a summary of meta-data variables that are used in OSCS. Here, `TID` represents a thread ID, `Addr` represents a byte address for each memory location, and `OrderObj` represents each ordering synchronization object. `ThreadVClk[i][i]` represents Thread `i`'s local clock, which is incremented on each ordering synchronization operation.

3.2 Detailed Algorithm

This subsection describes the details of the necessary meta-data and operations for each step in the algorithm. Figure 8 shows the pseudo-code for operations on each memory access and each synchronization operation.

3.2.1 Critical Sections with Conflicting Accesses

The algorithm detects a critical section pair with conflicting accesses in two steps: detect conflicting accesses, and

Functions on each memory access:

```
OSCS_detectCS(TID, Addr, Type, ThreadVClk[TID][TID]):
1. Check the most recent write in each thread:
   (a) For all valid threadID i ≠ TID,
       (a.1) If (CSList[TID] ∩ PrevWriteCSList[Addr][i])
             Call check_order(TID, i,
             PrevWriteVClk[Addr][i], Addr).
2. Check the most recent read in each thread:
   (a) If (Type == Read), skip Step 3.
   (b) For all valid threadID i ≠ TID,
       (b.1) If (CSList[TID] ∩ PrevReadCSList[Addr][i])
             Call check_order(TID, i,
             PrevReadVClk[Addr][i], Addr).
3. Update the access history for the memory location
   (a) If (Type == Read),
       PrevReadVClk[Addr][TID] = ThreadVClk[TID][TID].
       PrevReadCSList[Addr][TID] = CSList[TID].
   (b) If (Type == Write),
       PrevWriteVClk[Addr][TID] = ThreadVClk[TID][TID].
       PrevWriteCSList[Addr][TID] = CSList[TID].

check_order(TID, PrevTID, PrevTimeStamp, Addr)
1. If (ThreadVClk[TID][PrevTID] ≤ PrevTimeStamp)
   Call check_commutative(TID, PrevTID, Addr).

update_commutative_flag(TID, Addr, Type):
1. If (Type == Write &&
      (CSList[TID] ∩ PrevReadCSList[Addr][TID])),
      CSRWFlag[Addr][TID] = True.
   Else, CSRWFlag[Addr][TID] = False.
2. If (Type == Read &&
      !(CSList[TID] ∩ PrevReadCSList[Addr][TID])),
      CSRWFlag[Addr][TID] = False.

check_commutative(TID, PrevTID, Addr)
1. If (!(CSRWFlag[Addr][PrevTID])),
   Report a bug.
2. If (CSRWFlag[Addr][PrevTID] && !(CSRWFlag[Addr][TID])),
   Add an entry to DetectionList[TID].
```

Functions on synchronization operations:

```
update_acquire(TID, SyncObj)
1. If (type(SyncObj) == OrderObj),
   (a) For each i, update the vector clock: ThreadVClk[TID][i] =
       MAX(ThreadVClk[TID][i], OrderObjVClk[SyncObj][i]).
   (b) ThreadVClk[TID][TID]++;
2. If (type(SyncObj) == MutexObj),
   Add SyncObj to CSList[TID].

update_release(TID, SyncObj)
1. If (type(SyncObj) == OrderObj),
   (a) For each i, update the vector clock: OrderObjVClk[TID][i] =
       MAX(ThreadVClk[TID][i], OrderObjVClk[SyncObj][i])
   (b) ThreadVClk[TID][TID]++;
2. If (type(SyncObj) == MutexObj),
   (a) Remove SyncObj from CSList[TID].
   (b) If (CSList[TID] is empty),
       For each access pair in DetectionList[TID],
       If (!(CSRWFlag[Addr][PrevTID]) || !(CSRWFlag[Addr][TID])),
       Report a bug.
```

Figure 8: The detailed OSCS algorithm.

check if they are within critical sections. To detect conflicting accesses, the algorithm maintains timestamps for the most recent read and write from each thread to each memory location, `PrevReadVClk[Addr][TID]` and `PrevWriteVClk[Addr][TID]`. The timestamps are from each thread's local clock. On each memory access, the algorithm updates the meta-data and checks if there is a conflicting access by looking up previous reads and writes from other threads.

To identify corresponding critical sections for conflicting accesses, the algorithm uses `CSList[TID]` and `PrevReadCSList/PrevWriteCSList[Addr][TID]`. `CSList` records the critical sections that each thread is currently running by recording the mutex object (`MutexObj`) and the timestamp of the corresponding lock() operation for each critical section. `PrevReadCSList` and `PrevWriteCSList` record the critical sections for the most recent read and write accesses to each memory location from each thread. The algorithm checks if conflicting accesses are both protected by a critical section with the same lock by comparing `CSList` and `PrevReadCSList/PrevWriteCSList`.

The algorithm updates `CSList` on each acquire or release operation. Updates for the rest of the meta-data and checks are performed on each memory access.

3.2.2 Ordered Critical Sections

Once a candidate critical section pair is identified, the algorithm checks if the pair is constrained to run in a deterministic order. For this purpose, the algorithm uses vector clocks per thread (`ThreadVClk[TID1][TID2]`) and ordering synchronization object (`OrderObjVClk[OrderObj][TID]` to encode ordering constraints between threads. `ThreadVClk[i][j]` shows the earliest that a memory access from Thread `i` can be executed in terms of Thread `j`'s local time without violating the ordering constraint from synchronization. The `OrderObjVClk` represents the earliest that the following acquire operation can happen in each thread's local time for each ordering synchronization object. `ThreadVClk[i][i]` represents Thread `i`'s local clock, which is incremented on each ordering synchronization operation.

On an *acquire* operation by Thread `i` for ordering synchronization, the `ThreadVClk[i]` is updated with `OrderObjVClk[OrderObj]` by taking the larger timestamp for each element. On a *release* operation for ordering synchronization, `OrderObjVClk` is updated with `ThreadVClk`. The operations are shown in `acquire_update()` and `release_update()`.

The algorithm uses the vector clocks to check if conflicting accesses from two critical sections are ordered (shown in `check_order()`). The current memory access from Thread `TID` and a previous access from Thread `PrevTID` are not explicitly ordered if `PrevVClk[Addr][PrevTID]` is greater or equal to `ThreadVClk[TID][PrevTID]`.

3.2.3 Commutative Critical Sections

To detect commutative critical sections, the algorithm uses a 1-bit flag (`CSRWFlag[Addr][TID]`) to indicate whether a *read-write* sequence within a critical section has happened to each memory location from each thread. On a write, `CSRWFlag` is set when the most recent local read was also within the same critical section, and cleared otherwise. The algorithm also clears `CSRWFlag` on the first read within a critical section or reads outside critical sections. These updates are shown in `update_commutative_flag()`.

In our heuristic, a critical section pair is considered commutative if *both* critical sections contain a read-write sequence for each conflicting memory location. The algorithm checks this property by calling `check_commutative()` on detecting unordered conflicting accesses from two critical sections. If `CSRWFlag` is not set for the earlier conflicting access, the algorithm can immediately determine that critical sections are non-commutative. However, if the previous access has `CSRWFlag` set while the current access does not, the conflicting access pair is added to `DetectionList` and checked again when the current thread exits the critical section (shown in `release_update()`).

3.2.4 Redundant Writes

The algorithm can also check memory values before and after a conflicting write to see if the write is redundant. We

discuss its impact on false positives in the evaluation section. For simplicity, this check is not shown in the pseudo-code.

3.3 Debug Information on Detection

The OSCS scheme can pinpoint the critical sections and conflicting memory accesses for a potential bug. Our implementation reports the program counters and memory addresses for conflicting accesses, the associated thread IDs, and the time-stamped mutex object for a potential bug to help debugging.

3.4 Optimizations

For simplicity, we first described the OSCS algorithm using vector variables for each memory location. However, the per-thread, per-byte meta-data variables result in a significant memory overhead, which increases linearly with the number of threads. To be more practical, here we propose an optimized algorithm, named `OSCS-Opt`. We will refer to the original algorithm as `OSCS-Base`.

To reduce overhead and improve scalability, `OSCS-Opt` uses scalar variables in place of vector variables for bookkeeping. Specifically, for each location, the algorithm only maintains the history of the two most recent reads and two most recent writes from different threads rather than one read and one write per thread. The algorithm needs the most recent read and write from the current thread within a critical section to update `CSRWFlag`, and at least one recent read/write from another thread to detect conflicting accesses. The scalar variables do not increase with the number of threads.

`OSCS-Opt` further reduces the amount of meta-data on previous accesses by keeping the history only for accesses within critical sections to shared memory locations that have conflicting accesses, exploiting that detection only uses those accesses. Our test shows that shared memory locations are only a small fraction of the entire memory space. Recent studies [4, 8] also made the same observation.

Our evaluation results indicate that these optimizations significantly reduce the space overhead without a noticeable impact on detection capability.

4. HARDWARE IMPLEMENTATION

This section introduces a hardware implementation of the OSCS algorithm, named `OSCS-HW`. Figure 9 shows the high-level block diagram for the architecture where the blue (dark) blocks indicate the new components that are needed to support OSCS. The architecture is based on the `OSCS-Opt` algorithm, which uses scalar meta-data for each shared memory location, but further limits the meta-data for efficiency. Specifically, `OSCS-HW` detects shared locations only through on-chip caches by tagging cache blocks and maintains per-location meta-data only in an on-chip buffer, named AHB. The checker module performs bookkeeping and check operations at each core and maintains per-thread meta-data.

4.1 Extension for Shared Location Detection

In our architecture, each block in data caches has a 1-bit tag, which indicates whether the block is shared. The shared bit is set when a cache coherence event indicates that multiple cores access the same cache block with at least one write. More specifically, the shared bit is set when there is a downgrade request that changes the cache block state to either shared or invalid. For example, in a MESI protocol, the shared bit is set on the following requests: `M->S`, `M->I`, `E->S`, `E->I`, `S->I`, etc. This shared bit follows the data on-chip; a shared bit is written back to a lower-level cache, and the bit is read with data on a cache miss. However, the shared bit is cleared when a cache block is evicted to or read from off-chip memory. Effectively, this mechanism detects memory blocks that are *shared* within a time window while the block exists in multiple private (L1/L2) caches, and keeps this history while the block is on-chip. While this design can only detect shared locations within a short period, our experimental results show that the on-chip bookkeeping is sufficient for virtually all concurrency bugs tested.

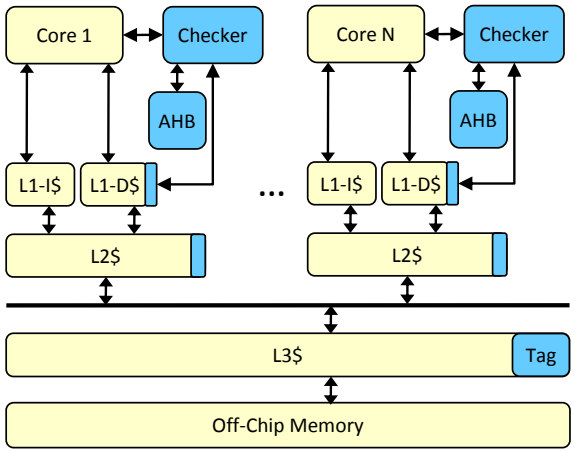


Figure 9: A block diagram for the overall architecture. Blue (dark) blocks are additional hardware needed for OSCS.

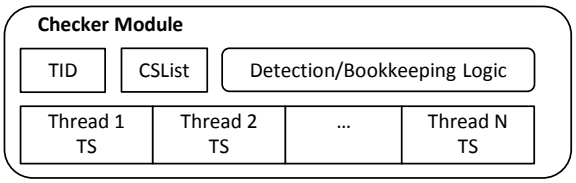


Figure 10: The checker module contains the thread ID (TID), the thread vector clock (ThreadVClk), the current critical section (CSList), and detection/bookkeeping logic.

We believe that the 1-bit tag is sufficient under the assumption that each core runs one thread with infrequent context switches or thread migrations. If not, the 1-bit tag may not detect locations that are shared by multiple threads on one core or incorrectly identify a block as shared when a single thread moves from one core to another. To be more accurate, a thread ID can be added to the 1-bit tag in order to identify which thread each access comes from. The overhead would be still quite low as only one thread ID needs to be maintained per cache block.

4.2 Checker Module

The checker module for `OSCS-HW` maintains a per-thread state and performs the bookkeeping and checking operations at each core. As shown in Figure 10, for the active thread on the core, the checker keeps a thread ID (`TID`), a thread vector clock (`ThreadVClk`), and the critical section that a thread is currently in (`CSList`).

We note that in order to support nested critical sections, we may need to store more than one critical sections in the `CSList`. However, we found that it is very rare to have nested critical sections in practice. In `OSCS-HW`, we made a

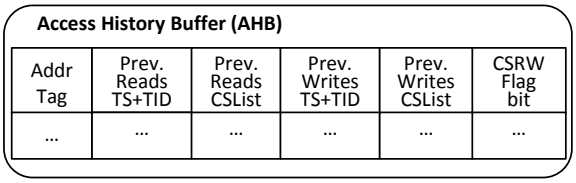


Figure 11: AHB: a history table that saves the information on the two most recent read and write accesses to shared memory locations within critical sections.

design choice to only maintain the most recent critical section ID and its timestamp. If necessary, a simple extension can be made by adding additional on-chip storage to store information on multiple nested critical sections.

On each memory access, the checker module checks if the memory location is "shared" and within a critical section by looking at a cache tag and `CSList`, and if so stores the access history in an on-chip buffer (AHB). The checker module also detects conflicting accesses within critical sections using the history in the AHB and the `CSList`, and performs checks for bug detection on those accesses.

For ordering synchronization operation, the checker module coordinates with a software layer through new instructions for bookkeeping. The architecture provides two additional instructions, one for *acquire* and the other for *release*, conveying the address of the vector clock for the synchronization object. The vector clock is accessed and/or updated by the checker module through normal memory hierarchy.

4.3 Access History Buffer

As shown in Figure 11, the access history buffer (AHB) in `OSCS-HW` keeps the meta-data for two most recent read and write accesses from different threads, including the TIDs, the timestamp, and the critical section information for those accesses. Additionally, each AHB entry also keeps the CSRWFlag for each location.

The AHB is kept coherent by a cache coherence protocol similar to other on-chip caches. We implemented the AHB coherence protocol separate from the main data cache, however, we note that as the total number of accesses to AHB is very small, the AHB coherence operations can also be implemented using the exciting main cache coherence structure with minimal overhead.

As the AHB has a limited capacity, it works like a cache and only keeps the history of recently accessed shared memory locations. However, there is no backup hierarchy for the AHB. If an entry is evicted from an AHB, the information is simply thrown away. A miss to the AHB creates a new entry. While this design implies that we cannot detect conflicting accesses with a long distance in between, our experimental results suggest that an AHB with 256 entries are sufficient for virtually all concurrency bugs tested. Moreover, a miss to the AHB can only lead to potential false negatives, but not false positives as we would not make a detection on a miss. We will further examine the impact of AHB size on the detection capability in our evaluation section.

We note that the AHB can store an access history *per byte* because only accesses to shared memory locations are recorded. On the other hand, traditional designs that combine meta-data into the main cache often had to store information on a cache block granularity to keep overheads acceptable. As an optimization, we have noticed that only a very small fraction of memory accesses were done on a byte granularity, while most of the memory accesses are to variables sized larger than a word. Therefore, we have implemented our AHB with flexible granularity. For each AHB entry, there is a single-bit flag which indicates if granularity is per byte or per word. An additional 4 bits are also appended to each entry to mark which bytes in a word are associate with the AHB entry. Overall, the flexible granularity improves our detection coverage by allowing more access histories to be kept in the AHB while still maintaining the per-byte detection granularity to avoid false positives.

4.4 Vector Clocks

Instead of keeping a vector clock per synchronization object, the OSCS scheme only requires a vector clock per ordering synchronization object. Therefore, the overall space and access overheads for vector clocks are insignificant in practice as the total number of ordering synchronization objects are generally small in applications. Moreover, as we only increment the vector clock elements on ordering synchronization operations, the hardware counters for the vector clocks would encounter an overflow very infrequently. In fact we did not encounter any overflows in our evaluation. Given that overflows are infrequent, our architecture handles them in a relatively slow but straightforward fashion instead of adding complex hardware. Upon detecting an overflow in its local clock, a checker raises an exception to an operating system, which in turn interrupts other cores that run other threads from the same program. Then, the operating system clears all clocks, and sets each thread's local time to one to prevent any false positives. In order to allow an operating system to clear vector clocks for synchronization objects, an application allocates them in separate pages that are known to the operating system.

5. EVALUATION

This section presents evaluation results for the proposed OSCS scheme in both software and hardware implementations. We first study the scheme's bug detection capability, then discuss the overheads.

5.1 Evaluation Setup

For the evaluation, we implemented three OSCS schemes, `OSCS-Base`, `OSCS-Opt`, and `OSCS-HW`, using the Pin binary instrumentation framework [15]. Our Pin tool implements the OSCS algorithms by intercepting memory accesses and Pthread calls.

To evaluate the architectural support, we implemented a typical memory hierarchy with bookkeeping structures in a Pin tool, and added a timing model with a processing core that runs 1 instruction per cycle, L1/L2/L3 caches, and a memory interface. Table 1 summarizes the baseline architecture parameters for `OSCS-HW`. For the OSCS bookkeeping, we model a 1-bit shared bit per cache block and a 256-entry AHB. An AHB entry records an 8-bit thread ID and a 16-bit timestamp per access, for two reads and two writes, and a 16-bit critical section ID and a 16-bit timestamp for the corresponding critical section.

To evaluate the bug detection capability, we used both real-world bugs and injected bugs. For real-world bugs, we first used five non-race bugs on two large real-world applications (MySQL and Mozzila). We also created kernel bugs (KB), which use two threads to reproduce real-world bugs (MySQL and Mozilla) that were reported in previous studies

Component	Parameters
Core	4 2-GHz in-order single-issue cores
Caches	L1 I/D (private, inclusive): 32KB/32KB 4-ways 3 cycle latency L2 (private): 256KB, 4-ways 15 cycle latency L3 (shared): 8MB, 8-ways 40 cycle latency
Coh. protocol	MESI
DRAM	4GB 50ns latency
Meta-data	8-bit thread IDs, 16-bit clocks
AHB	256-entries, 8-way, 9KB, 3 cycles latency

Table 1: Baseline architecture parameters.

Bugs	Description
MySQL1-A	Log status can be read intermediately from a remote thread while its two update operations are not within the same critical section.
MySQL2-AM	The log and table operations are not updated within the same critical section. The operation and its log may not match.
Mozilla1-A	NULL can be set in between script-set and script-compile.
Mozilla2-AM	Variable is being cleared in between initialization and setting the empty flag to false, resulting a mismatch.
Mozilla3-O	Loop break flag is set too early, resulting in an infinite loop.
KB1-A (MySQL)	After an update, the variable can be set back to zero intermediately by a remote thread.
KB2-AM (MySQL)	An updated variable and its logged updated value may not match due to remote intermediate update operations
KB3-O (Mozilla)	A waiting flag is checked before it's set, program hangs.
KB4-AM (Mozilla)	Two related variables are separately protected by different locks, and are not updated atomically together.

Table 2: Non-race concurrency bugs tested (A - Atomicity violation, O - Order violation, M - Multi-variable bug).

[10, 11, 24]. The list includes all real-world non-race bugs that we could find and reproduce. For a further study on detection, we also tested non-race bugs that are intentionally injected to programs from the SPLASH2 [3] and PARSEC [1] benchmark suites. The following subsection describes the details of the bug injection study.

For a false positive study, we ran the SPLASH2 (4 threads, default input size) and PARSEC (4 threads, `simmedium` input size) benchmarks. We also tested full deployed instances of Apache (30 threads), Mozilla (8 threads), MySQL (10 threads), Aget (8 threads), and Pbzip2 (8 threads). The workloads for them were created to mimic real uses.

5.2 Bug Detection Capability

Table 2 shows the real-world bugs that we tested. `OSCS-Base`, `OSCS-Opt` and `OSCS-HW` detect all real-world bugs in the table after a single run of each application, showing that the notion of order-sensitive critical sections is effective in practice. While our algorithm is not designed for multi-variable bugs, the results show that some multi-variable bugs can be detected from non-determinism in a single variable.

In order to further study the detection capability, we injected atomicity and ordering violations into benchmarks. For atomicity violation, we first created 5 bugs by manually selecting and breaking up a critical section into two or more: 2 in Ocean, 2 in Water-Nsquared, 1 in LU. For ordering violation, we manually removed a barrier or a wait instruction in a benchmark, creating 4 bugs: 2 in Ocean, 1 in Water-Nsquared, 1 in LU. Except for one injected ordering bug in LU, the OSCS algorithms successfully detected all manually injected bugs after a single run. The manually injected bugs were checked to ensure that they indeed change the program outcome or crash the program.

	Unique Cases			
	Injected	Detected by OSCS		
		Base	Opt	HW
Bodytrack(P)	3(A)	2(A)	1(A)	1 (A)
Fluidanimate(P)	5(A)	5(A)	5(A)	5(A)
LU	8(A)	6(A)	6(A)	5(A)
(S)	1(O)	0(O)	0(O)	0(O)
Ocean	14(A)	12(A)	12(A)	11(A)
(S)	2(O)	2(O)	2(O)	2(O)
Radix(S)	4(A)	4(A)	4(A)	4(A)
Water-Nsquared	16(A)	16(A)	16(A)	14(A)
(S)	1(O)	1(O)	1(O)	1(O)
Water-Spacial(S)	14(A)	14(A)	14(A)	14(A)
Total	68	62	61	57
Detection %	—	91%	90%	84%

Table 3: Detection of injected bugs (P - PARSEC, S - SPLASH2, A - Atomicity violation, O - Order violation).

To obtain more test cases, we performed an automatic injection of atomicity violation bugs using Pin. Here, the tool broke up one critical section into two by injecting an unlock/lock pair between shared memory accesses. The approach created a number of unique bug cases by sweeping all possible options. Note that the automatically injected bugs may not truly affect the program outcome; we could not check each case due to the large number.

Table 3 shows the detection results for both manually and automatically injected bugs combined. The results show that the proposed scheme can detect most of injected bugs; `OSCS-Base` detected 91%, `OSCS-Opt` detected 90%, and `OSCS-HW` detected 84%. Most of the injected bugs were consistently detected after a single run. Some bugs were only detected on certain program runs: 12 for `OSCS-base`, 15 for `OSCS-Opt`, and 18 for `OSCS-HW`. More than a half of these bugs were detected on more than two third of runs, and others were detected on about one third of runs.

`OSCS-Base` can detect bugs that `OSCS-Opt` cannot because `OSCS-Base` keeps track of the most recent read and write per thread whereas `OSCS-Opt` only uses two reads and two writes to detect conflicting accesses. Similarly, `OSCS-HW` has a lower coverage than the software schemes due to its limited bookkeeping space in hardware. However, results for both real-world bugs and injected bugs suggest that the proposed optimizations have a minimal impact on the bug coverage in practice. This is because most concurrency bugs involve memory accesses that are close to each other. For example, a previous study reported that typical atomicity violations happens within 750 accesses [14].

The detection capability of `OSCS-HW` depends on the data cache sizes as well as the AHB size because they determine the amount of meta-data that can be kept. We found that the detection coverage for injected bugs drops to to 76% and 60% respectively if the cache sizes are reduced to 1/2 (i.e. 16KB L1, 128KB L2, 4MB L3) and 1/4 (i.e. 8KB L1, 64KB L2, 2MB L3) of the baseline. On the other hand, increasing the cache sizes did not show any improvement in detection coverage. Similarly, we found that the detection coverage drops to 78% and 72% when the AHB size is reduced to 128 and 64 entries, respectively. Increasing the AHB size beyond the baseline (256 entries) did not show any change in detection capability.

	Apache	Bodytrack	Ocean	Pbzip2	Radix
OSCS -Base	4 (1 re-wr)	3	1 (2 re-wr)	6	1
OSCS -Opt	4 (1 re-wr)	3	1 (2 re-wr)	6	1
OSCS -HW	4 (1 re-wr)	3	1 (2 re-wr)	6	1

Table 4: The number of false positives on each program (Re-wr: redundant writes).

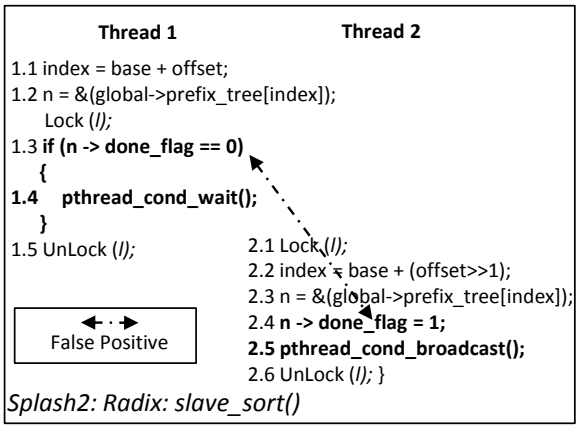


Figure 12: A false positive example from Radix.

5.3 False Positives

Table 4 shows false positives for our detection scheme. As shown, all three implementations have 17 false positives: 4 for Apache, 3 for Bodytrack, 1 for Ocean, 6 for Pbzip2 and 3 for Radix. Out of these false positives, 1 case in Apache and 2 cases in Pbzip2 were redundant writes that can be avoided by checking values for writes. There was no false positive for other programs including Aget, MySQL, Mozzila, PARSEC, and SPLASH2 benchmarks.

Four false positives from Pbzip2 and all false positives from Apache, Bodytrack, and Radix come from a synchronization operation that uses a `pthread_cond_wait()`. Figure 12 shows an example from Radix. Here, the `done_flag` in the global variable `prefix_tree` is set in Thread 2 while Thread 1 is waiting for the broadcast signal. In effect, the shared variable in this case is used as a flag variable. These false positives are similar to spin flags that can cause false positives in data race detectors and can be avoided if the flag variable can be identified as an ordering synchronization.

Other false positives come from the use of heuristic in identifying commutative critical sections. For example, the false positive in Ocean happens because a MAX function only updates a global variable when it is less than the local value (discussed earlier in Figure 5). Two false positives from Pbzip2 involve head and tail pointers of a consumer-producer FIFO queue as shown in Figure 13. Here, the head and tail pointers are read by both the producer and consumer, but incremented only by one. In this case, high-level programmer knowledge is needed to understand that non-determinism in the head and tail pointers only affect timing but not program state.

Overall, the false positive study shows that the proposed algorithm indeed has false positives, but only has a small number of them in practice. Given that the proposed heuristic can detect a broad range of non-race bugs, we believe that the small number of false positives are acceptable. In fact,

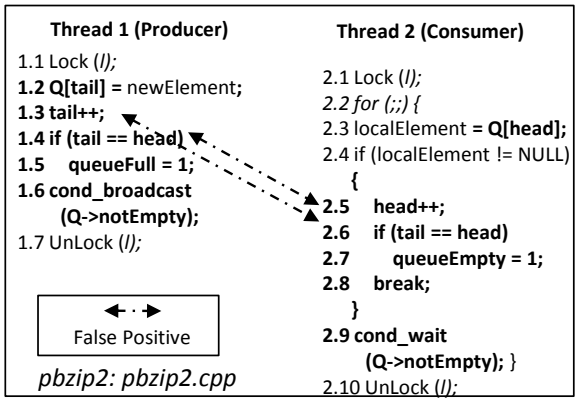


Figure 13: A false positive example from Pbzip2.

	Pin slowdown	OSCS-Base slowdown over Pin	OSCS-Opt slowdown over Pin
Bodytrack (P)	11.25x	25.43x	11.75x
Fluidanimate (P)	12.16x	31.46x	22.32x
LU (S)	14.10x	32.85x	22.78x
Ocean (S)	13.43x	34.53x	23.65x
Radix (S)	21.11x	32.15x	22.77x
Water-Nsquared (S)	12.52x	31.63x	21.18x
Water-Spacial (S)	13.93x	31.48x	20.79x
Geomean	13.80x	31.24x	21.52x

Table 5: SW performance overhead. The slowdowns are normalized to the execution times of Pin instrumentation.

the number of false positives in our experiments is comparable to the number of false positives (data races that are not bugs) in happens-before data race detectors.

5.4 Performance Overheads

Table 5 shows the performance of our software implementations. For our benchmarks, the Pin framework without any instrumentation incurs a slowdown of 13.80x on average. In addition to the overhead of Pin itself, our detection algorithms show an average slowdown of 31.24x for `OSCS-Base` and 21.52x for `OSCS-Opt`. `OSCS-Opt` is noticeably faster thanks to its smaller memory footprint and the fact that it only checks up to two most recent accesses in detecting conflicting accesses. The performance overhead is comparable to a race detector based on vector clocks that we implemented in Pin as a comparison. Also, we believe that the software implementations can be further optimized.

Figure 14 shows the normalized execution time for `OSCS-HW`. The performance overhead is almost negligible. In the worst case, the overhead is 0.5% for `Fluidanimate`. Because the architecture mostly uses dedicated on-chip structures such as 1-bit cache tags and the AHB for bookkeeping, there are only two main sources of overhead for our implementation: the coherence traffic among AHBs and vector clock accesses through the normal memory hierarchy for ordering synchronization objects. However, because the AHB is only accessed on a shared memory access within a critical section, we found that AHB accesses are rather infrequent (1.7% in the worst case for `Fluidanimate` and 0.23% on average). Similarly, the number of vector clock accesses are negligible when compared to the number of regular data accesses (0.07% in the worst case for `Fluidanimate` and only 0.003% on average). While counter overflows may also introduce performance overhead by requiring timestamps and

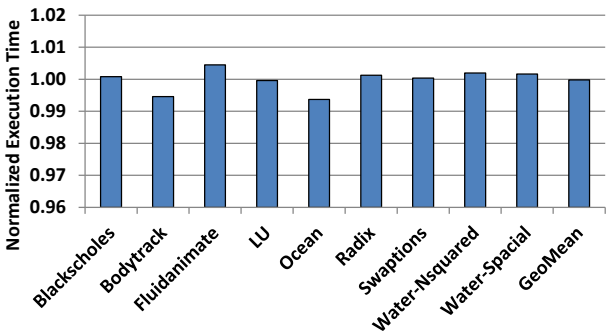


Figure 14: HW performance overhead over native execution.

vector clocks to reset, the timestamp values never exceeded a few hundreds after the entire execution for our benchmarks. As a result, the overall performance impact is minimal for `OSCS-HW`.

We note that in certain cases (`Bodytrack` and `LU`), the experiments show a tiny speedup for `OSCS-HW`. This is because additional vector clock accesses change the cache access pattern, which resulted in a slightly lower cache miss-rate. The speedup is quite small and less than 0.5%.

5.5 Memory Space Overhead

The memory space overhead of the software schemes mainly comes from the per-address per-thread meta-data such as `PrevCSList` and `PrevVClk`. Specifically, our scheme maintains a 32-bit critical section ID (16-bit mutex object ID and 16-bit timestamp for lock()), and a 16-bit timestamp for the most recent read and write to each location, which represents a 12-byte overhead per thread per byte for a byte-addressable system. For `OSCS-Base` with 8 threads, the space overhead is 96 times of the original program's memory footprint. On the other hand, `OSCS-Opt` has a much lower memory space usage thanks to the optimization that only keeps the history of only two recent accesses for shared memory locations. We found that the memory space usage is roughly 2-3x. `OSCS-HW` has no per-address overhead because the meta-data is only kept in the AHB.

In addition to the per-address meta-data, all three OSCS schemes require a vector clock for each thread and each ordering synchronization object. Each vector clock has N_{thread} timestamps where N_{thread} is the number of threads. However, the space overhead of these vector clocks is rather small because the number of threads and the number of ordering synchronization objects are small. In our experiments, the number of ordering objects never exceeded 10. Even with 100 threads and 100 ordering synchronization objects, the vector clocks only require 2MB for 16-bit timestamps.

5.6 Comparison to Other Schemes

Table 6 compares the characteristics of the OSCS scheme with three state-of-the-art non-race bug detection schemes. All four schemes are based on heuristics and as a result have false positives and negatives.

SVD [23] and AVIO [11] both detect atomicity violations by approximating intended atomic regions from common program behaviors. AVIO has a high coverage in detecting atomicity violations, but requires training runs. On the other hand, SVD has a lower coverage, only detecting 1 out of 6 bugs studied in AVIO [11]. SVD and AVIO cannot detect ordering violations or bugs with multiple variables. The

	SVD	AVIO	Bugaboo	OSCS
Training Runs	No	Yes	Yes	**No**
Detect AV	Yes	Yes	Yes	**Yes**
Detect OV	No	No	Yes	**Yes**
Detect MV	No	No	Yes	**Yes***
Detection Coverage	Low-Med	High	High	**90%(SW) 84%(HW)**
False Negative	Yes	Yes	Yes	**Yes**
False Positive	Yes	Yes	Yes	**Yes**
SW Slowdown	65x	25x	15x-5025x	**300x**
HW Slowdown	N/A	NEGL	NEGL	**NEGL**

Table 6: Comparisons to other non-race bug detection schemes (AV - Atomicity Violation, OV - Ordering Violation, MV - Multivariable Violation, NEGL - negligible).

OSCS scheme detects all 6 out of 6 bugs studied by AVIO, without training, including a multi-variable bug that AVIO could not detect.

Bugaboo [12] provides a coverage for atomicity and ordering violations, and can detect both single and multi-variable bugs. However, it requires multiple training runs with explicit labeling of correct and incorrect runs to lower false positives. Even than, Bugaboo reports higher false positives than OSCS (1 true bug from 8 detections).

Overall, compared to the state-of-the-art, the proposed OSCS scheme provides high detection coverage for both atomicity and ordering violations with low false positives without requiring program-specific training runs

6. RELATED WORK

There exist many approaches on detecting concurrency bugs. Here, we discuss previous schemes that are most closely related to the proposed scheme.

The OSCS scheme detects non-race concurrency bugs by extending the intuition behind data races to critical sections. Also, the proposed run-time algorithm uses vector clocks in a similar fashion as traditional data race detection schemes use vector clocks [5, 18, 19]. However, using the new notion of order-sensitive critical sections, our scheme detects non-race bugs, which cannot be detected by any data race detector.

Recently, there have been significant efforts to detect concurrency bugs using symptoms beyond data races. One popular approach is to detect and/or tolerate bugs based on common program behaviors. For example, AVIO [11] and SVD [23] approximate intended atomic regions using common behaviors, Atom-Aid [14] tries to dynamically avoid atomicity violation bugs, MUVI [9] and ColorSafe [13] target to detect concurrency bugs that involve multiple variables, and Bugaboo [12] detect anomalies in communication graphs. Alternatively, researchers have also found that concurrency bugs can be identified from their consequences such as memory errors [26] or other failure patterns [25]. Traditionally, a program is often tested by running with many possible interleaving patterns and checking results [16, 17, 2]. This approach can detect any concurrency bug if a buggy interleaving is tried, yet can only test one case at a time.

This work presents a new approach to detect non-race bugs by introducing the concept of order-sensitive critical sections as a new indicator of a bug. This approach can be applied to programs without learning application-specific or bug-specific behaviors, and can often identify potential bugs before they happen at run-time. Moreover, the proposed scheme can detect multiple bug types including both atomicity and ordering violations as the OSCS heuristic identifies

the non-determinism in shared memory state, which can be caused by any type of violations. However, this technique is still a heuristic and there is no guarantee on bug detection coverage. In this sense, the proposed scheme complements an existing body of work in non-race bug detection.

7. CONCLUSION

This paper introduces the notion of order-sensitive critical sections, which captures critical sections that introduce non-determinism during executions, and presents a concurrency bug detection approach that covers non-race bugs. An optimized version of our algorithm is also introduced, which only requires scalar variables for each shared memory address instead of using vectors for all memory addresses. Experimental results show that a broad range of non-race concurrency bugs can be detected by both the baseline and the optimized algorithms. In particular, our algorithms detected all 9 real-world bugs and over 90% of all injected bugs with only a small number of false positives in practice. The algorithm can also be accelerated with hardware support to have minimal performance overhead. The hardware implementation can still detect all 9 real-world bugs and above 84% of all injected bugs. Overall, we demonstrate that the proposed OSCS scheme is indeed an effective heuristic for non-race concurrency bug detection, and it can be implemented in hardware effectively and efficiently.

8. ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation under grants CNS-0746913 and CCF-0905208, by the Air Force Office of Scientific Research under Grant FA9550-09-1-0131, and an equipment donation from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, AFOSR, or Intel.

9. REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the* 15^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[3] CAPSL. The modified SPLASH-2. `http://www.capsl.udel.edu/splash/`, July 2007.

[4] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the* 38^{th} *Annual International Symposium on Computer Architecture*, 2011.

[5] J. Deviett, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. Radish: always-on sound and complete race detection in software and hardware. In *Proceedings of the* 39^{th} *Annual International Symposium on Computer Architecture*, 2012.

[6] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:28–33, August 1991.

[7] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture*, 2009.

[9] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of* 21^{st} *ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the* 13^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the* 12^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[12] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the* 42^{nd} *Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[13] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the* 37^{th} *Annual International Symposium on Computer Architecture*, 2010.

[14] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: detecting and surviving atomicity violations. In *Proceedings of the* 35^{th} *Annual International Symposium on Computer Architecture*, 2008.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the* 2005 *Conference on Programming Language Design and Implementation International*, 2005.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the* 8^{th} *USENIX Conference on Operating Systems Design and Implementation*, 2008.

[17] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceeding of the* 14^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[18] M. Prvulovic. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the* 12^{th} *Annual International Symposium on High-Performance Computer Architecture*, 2006.

[19] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the* 30^{th} *Annual International Symposium on Computer Architecture*, 2003.

[20] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008.

[21] C. Valot. Characterizing the accuracy of distributed timestamps. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[22] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the* 9^{th} *USENIX Conference on Operating Systems Design and Implementation*, 2010.

[23] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[24] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture*, 2009.

[25] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Proceedings of the* 16^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[26] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the* 15^{th} *Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.